Two Dimensional Parallelism using DistributedTensor

Wanchao Liang, Junjie Wang, PyTorch Distributed, Dec 2022

Outline

- Background and Motivation
- DistributedTensor Introduction
- DistributedTensor API
- Tensor Parallelism and user API
- 2-D Parallelism (Tensor Parallelism + Data Parallelism)
- Preliminary result on ViT Transformer

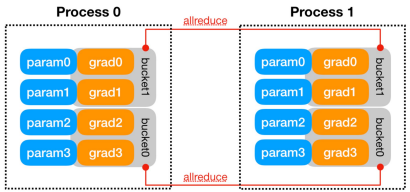
Background & Motivation

Distributed Training in research community

currently research community train large scale models mainly in three ways:

1. Data Parallel

partitions and train data in parallel, model is usually replicated across devices



PyTorch:
Distributed Data Parallel
Fully Sharded Data Parallel

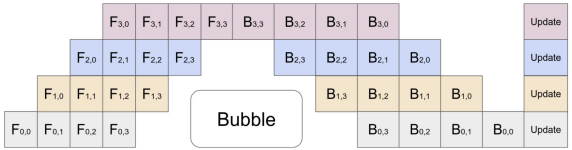
2. Within-layer Model Parallel

i.e. Tensor Parallel that partition weights and computation

PyTorch:
DistributedTensor (Tensor Parallel)

3. Pipeline Parallel

partitions the computation graph and run pipeline across stages



(Diagram from Huang, 2018)

PyTorch:
PiPPy (prototype stage)

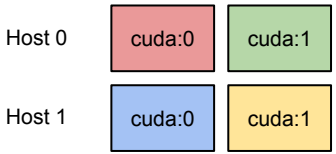
DistributedTensor

A common abstraction that build the bridge between different parallelism strategies.

DistributedTensor (a.k.a DTensor) offers fundamental tensor primitives to describe data distributions and computation across hosts.

DeviceMesh:

An abstraction that describes global view/topology of devices within a cluster. i.e. a 2-D mesh [[0, 1], [2, 3]]

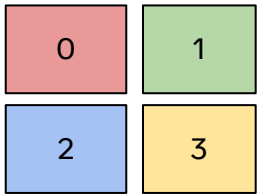


Distributed Tensor Types

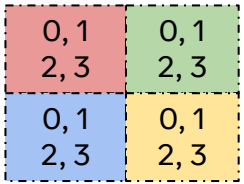
- ***Shard:*** *shard on tensor dimension across devices*
- **Replicate**: *replicate across devices.*
- ***Partial***: *partition values across devices.*

Data to distribute:

[0, 1],
[2, 3]

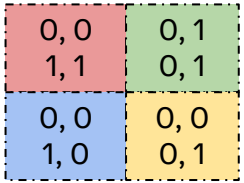


Shard
Spec: [Shard(0), Shard(1)]



Replicate
Spec: [Replicate(), Replicate()]

Replicate + Shard
Spec: [Shard(0), Replicate()]



Partial
Spec: [Partial(), Partial()]

DistributedTensor API

DTensor API Examples

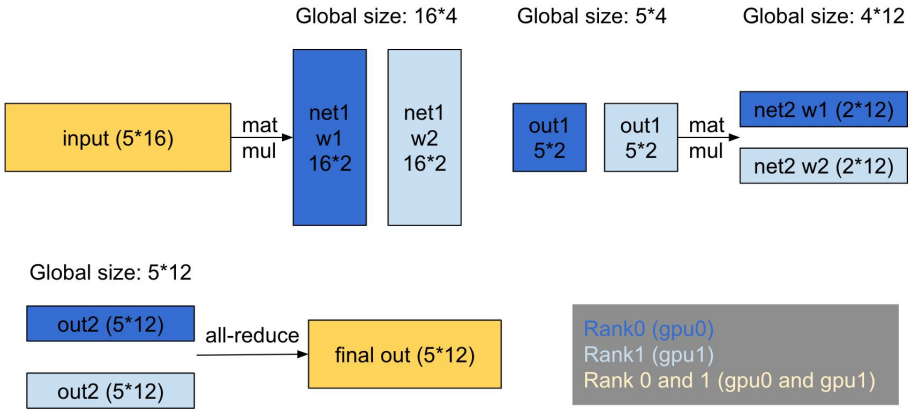
```
device_mesh = DeviceMesh(device_type="cuda", mesh=[0, 1, 2, 3])

# create a DTensor from local tensor (shard) on each rank
local_tensor = torch.randn((888, 1888)
colwise_dt = DTensor.from_local(local_tensor, device_mesh, [Shard(1)])
# redistribute the DTensor from col-wise to row-wise
rowwise_dt = colwise_dt.redistribute(device_mesh, [Shard(0)])
# redistribute the DTensor from row-wise to replicate
replica_dt = rowwise_dt.redistribute(device_mesh, [Replicate()])
# convert colwise_dt back to local tensor (shard)
local_tensor = col_wise_dt.to_local()

# shard a existing Tensor from one rank to all ranks
sharded_dt = distribute_tensor(torch.randn(8888), device_mesh, [Shard(0)])
```

DTensor API details

- DTensor is a ***torch.Tensor subclass***.
- Convert from/to local torch.Tensor via *from_local* and *to_local*
- Reshard/Redistribute a DTensor from one sharding to another
- Offering high level functional APIs that transforms a tensor/module to distributed tensor/module
  - ***distribute_tensor(tensor)***
  - ***distribute_module(module)***

Tensor Parallelism and user API

- colwise parallelism:
  - Partition dim -1 of weight/bias matrix
- rowwise parallelism:
  - Partition dim 0 of weight/bias matrix
- pairwise parallelism:
  - Cascate a colwise with a rowwise as a pair



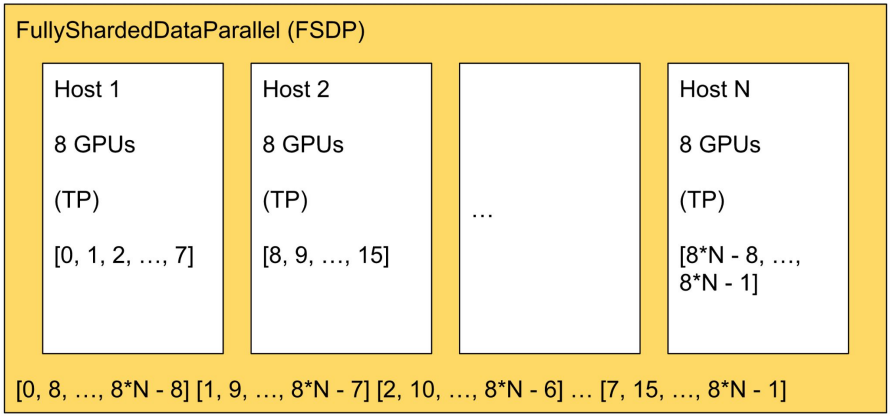
Tensor Parallel API Examples

```
# initialize a new device mesh for Tensor Parallel
device_mesh = DeviceMesh("cuda", torch.arange(ws))
# colwise parallel of a Linear module
m = torch.nn.Linear(8,16)
parallelize_module(
  m,
  ColwiseParallel(),
  device_mesh,
)

# pairwise parallel style for a transformer model
customized_model = DemoModel(...)
pairwise_style = PairwiseParallelStyle()
parallelize_module(
  customized_model,
  {"attn": pairwise_style, "mlp": pairwise_style},
  device_mesh,
)
```

Two Dimensional Parallelism (Tensor Parallelism + Data Parallelism)

We enabled Fully Sharded Data Parallel + Tensor Parallel in separate parallel dimensions:

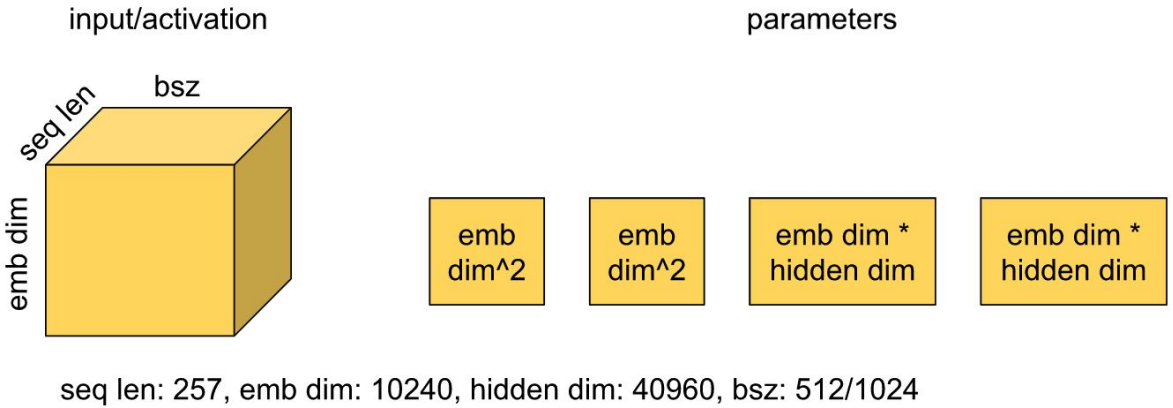
- Data Parallel across hosts
- Tensor Parallel within each host




```
# initialize a new device mesh for 2D parallel for the given world size
device_mesh = DeviceMesh("cuda", torch.arange(ws).reshape(dp_size, tp_size))
# Pairwise sharding for a transformer model
parallelize_module(model, pairwise_style, device_mesh_2D, tp_mesh_dim=1)
# Wrap the transformer with FSDP
dp_pg = device_mesh.get_dim_groups()[0]
transformer_model = FSDP(transformer_model, pg=dp_pg)
```

Preliminary results on Vision Transformer model

- 2D parallelism vs FSDP only on 64 GPUs:
  - 2.6 times faster in training 60B Vision Transformer (ViT)
  - Enabled the training of 120B ViT (vs. OOM)



Quantitative analysis for peak memory for one layer:

- bsz = 1024, FSDP use 1.3 times more memory, which is 1.1 in experiment
- bsz = 512, the ratio increases to 2.4

Thank you!